18-100 Lecture 20: 
AVR Programming, Continued

James C. Hoe
Dept of ECE, CMU
April 2, 2015

Today’s Goal: You will all be ace AVR hackers!

Announcements: Midterm 2 can be picked up in lab and at the HUB
Office Hours: Wed 12:30~2:30

Handouts: HW8 (on Blackboard, due next Thursday)
Lab10 (on Blackboard)

“AVR” Program Visible State
(ones we care about for now)

Program Memory

Data Memory

4K (2^12) 16-bit words
-1 instruction
1 or 2 words

16-bit PC
1K (2^10) 8-bit words
-between 0x060~0x45F
-what’s in 0x000~0x05F?

8-b status

Register File

32 8-bit words
AVR Data Types

- The 8-bit data word (in register or memory) can represent
  - a collection of 8 bits
  - an “unsigned” integer value between 0 and 255
  - a “signed” integer value between -128 and 127

Questions
- how do you represent integer values as bits?
- what if I want to represent a value greater than 255?
- why -128 but only 127 in option 3?
- if I showed you an 8-bit data word in a register, how do you know which interpretation to apply?
- what about fractions and real numbers?

Representing Unsigned Integers

- How to represent 0~255 with 8-bits
  - 8 bits has $2^8 = 256$ different patterns of eight 1s and 0s
  - any one-to-one assignment of integer values to patterns is “correct”

- A conventional mapping based on counting is convenient in most cases
  - $00000000_2$, $00000001_2$, $00000010_2$, $00000011_2$, $00000100_2$, $00000101_2$, $00000110_2$, $00000111_2$
  - $00001000_2$, $00001001_2$, $00001100_2$, $00001101_2$, $00001110_2$, $00001111_2$
  - $\ldots$, $9_{10}$, $25_{10}$, $\ldots$ $1111011_2$, $1111100_2$, $1111101_2$, $1111110_2$, $1111111_2$
Intuition: a 4-bit example

- What it means to “carry”?

Unsigned Integer Representation

- In general, let \( b_{n-1}b_{n-2}...b_2b_1b_0 \) represent an n-bit unsigned integer
  - its value is \( \sum_{i=0}^{n-1} 2^i b_i \)
  - a finite representation between 0 and \( 2^n - 1 \)
  - e.g., \( 1011_2 = 8 + 2 + 1 = 11_{10} \)

- Often written in hexadecimal for compactness
  - to convert, starting from the right, map 4 binary digits at a time into a corresponding hex digit \( \{0-9, a-f\} \); and vice versa
  - e.g., \( 1010_2\text{110}_2 = ab_{16} \)
Representing Signed Integers

- How to represent -128~127 with 8-bits
  - any one-to-one assignment of integer values to the 256 patterns is “correct”
  - probably makes sense that 0_{10} to 127_{10} should be 00000000_{2} to 01111111_{2}
- What about -1 to -128?
- The logical extension is to count down from 0,
  \[-10=11111111_{2}; \quad -2=11111110_{2}; \quad -3_{10}=11111111_{2} ;\]
  \[-4_{10}=11111100_{2}; \quad -5_{10}=11111101_{2}; \quad -6_{10}=11111011_{2} ;\]
  \[\ldots\]
  \[-126_{10}=10000010_{2}; \quad -127_{10}=10000001_{2}; \quad -128_{10}=10000000_{2}\]
  This scheme is called “2’s complement”

2’s Intuition: a 4-bit example

- How does AVR’s ADD know it is adding 2’s or unsigned?
- What it means to overflow?
2’s-Complement Number Representation

- Let $b_{n-1}b_{n-2}...b_2b_1b_0$ represent an $n$-bit signed integer
  - its value is $-2^{n-1}b_{n-1} + \sum_{i=0}^{n-2}2^ib_i$
  - a finite representation between $-2^{n-1}$ and $2^{n-1}-1$
  - e.g., assume 4-bit 2’s-complement
    $b’1011 = -8 + 2 + 1 = -5$
    $b’1111 = -8 + 4 + 2 + 1 = -1$

- To negate a 2’s-complement number
  - add 1 to the bit-wise complement
  - assume 4-bit 2’s-complement
    (- $b’1011) = b’0100 + 1 = b’0101 = 5$
    (- $b’0101) = b’1010 + 1 = b’1011 = -5$
    (- $b’1111) = b’0000 + 1 = b’0001 = 1$
    (- $b’0000) = b’1111 + 1 = b’0000 = 0$

Status Register

- ‘V’, ‘N’, ‘Z’, ‘C’ are arithmetic flags automatically updated after each ALU-class instructions
  - Z: set if the last result was zero,
  - N: set if the last result was negative (2’s complement)
  - V: set if the last op caused an overflow (2’s comp)
  - C: set if the last op caused a carry (unsigned)
What about larger integers

- A 16-bit integer can be held using two 8-bit registers
- Add/sub need to be emulated using multiple native 8-bit operations
- Suppose 16-bit A’s upper/lower bytes are in r17/r16 and 16-bit B’s upper/lower bytes are in r19/r18

\[
\begin{align*}
\text{add } r16, r18 &; \text{ sets C flag if carry} \\
\text{adc } r17, r19 &; \text{ incorporates C flag in sum}
\end{align*}
\]

Extensible to larger integers but commensurately more expensive

General Instruction Classes

- Arithmetic and logical operations
  - fetch operands from specified locations
  - compute a result as a function of the operands
  - store result to a specified location
  - update PC to the next sequential instruction
- Data movement operations
  - fetch operands from specified locations
  - store operand values to specified locations
  - update PC to the next sequential instruction
- Control flow operations
  - fetch operands from specified locations
  - compute a branch condition and a target address
  - if “branch condition is true” then PC ← target address
  - else PC ← next seq. instruction
Control Flow Instructions

**C-Code**

```c
{ code block A }
if X==Y then
    { code block B }
else
    { code block C }
{ code block D }
```

**Control Flow Graph**

- **Block A**
  - True branch
- **Block B**
- False branch
- **Block C**
- **Block D**

**Assembly Code (linearized)**

- **Code A**
  - if X==Y
- **Code B**
- **Code C**
- **Code D**

These things are called basic blocks.

Control Flow: Jump!

**RJMP – Relative Jump**

**Description:**

Relative jump to an address within PC + 2K + 1 and PC + 2K (words). For AVR microcontrollers, this instruction can address the entire memory from every address location. See also JMP.

**Operation:**

(i) \( PC \leftarrow PC + k + 1 \)

**Syntax:**

RJMP k

**Operands:**

\(-2K \leq k \leq 2K\)

**Program Counter:**

PC \leftarrow PC + k + 1

**16-bit Opcode:**

| 1100 | kkkk | kkkk | kkkk |

**Note:** Jump target is specified as an offset from PC+1, but, fortunately, in assembly programs, you can specify the label of an “absolute” target instruction and the assembler will figure out the offset. (example later)
Control Flow: Branch?

BREQ – Branch if Equal

Description:
Conditional relative branch. Tests the Zero Flag (Z) and branches relatively to PC if Z is set.
This instruction branches relatively to PC in either direction (PC - 63 ≤ destination ≤ PC + 64).
The parameter k is the offset from PC and is represented in two's complement form.

\[(Z = 1) \text{ then } PC \leftarrow PC + k + 1, \text{ else } PC \leftarrow PC + 1\]

Syntax: \(\text{BREQ } k\)  
Operands: \(-64 \leq k \leq +63\)

Program Counter:
\(PC \leftarrow PC + k + 1\)  
\(PC \leftarrow PC + 1\), if condition is false

16-bit Opcode:

Note:
- Like in RJMP, a branch target is also PC-relative
- \((Z=1)\) is the branch condition. What the heck is \(Z\)?

Status Register

don’t worry about 7~4

- ‘Z’, ‘N’, ‘V’, ‘C’ have corresponding branch instructions
  - BREQ/BRNE, BRMI/BRPL, BRVS/BRVC, BRCS/BRCC

- E.g.,
  - after “SUB Rx, Ry” or “CP Rx, Ry”, \(Z\) is set if \(Rx=Ry\)
  - BREQ taken if \(Rx=Ry\), BRNE taken if \(Rx!=Ry\)
Assembly Programming 201

- E.g. High-level Code
  ```java
  if (i == j) then
      e = g
  else
      e = h
  f = e
  ```

- Assembly Code
  - suppose e, f, g, h, i, j are in r_e, r_f, r_g, r_h, r_i, r_j
  ```assembly
  cp r_i, r_j ; set status flags
  brne L1 ; if i!=j skip to L1 (else)
  mov r_e, r_g ; e gets g
  rjmp L2 ; skip to L2 (join)
  L1: mov r_e, r_h ; e gets h
  L2: mov r_f, r_e ; f gets e
  . . .
  ```

Assembly Programming 202

- High-level Code
  ```java
  i=0; j=10;
  while (i != j) {
      i++;
  }
  . . .
  ```

- Give it try. Pick your own free registers
Assembly Programming 202

- High-level Code

```c
i=0; j=10;
while (i != j) {
  i++;
}
...
```

- Give it try. Pick your own free registers

```assembly
ldi r_i, 0 ; r_i = 0
ldi r_j, 10 ; r_j = 10
loop:
cp r_i, r_j ; set NZCV on r_i-r_j
breq done ; r_i==r_j then done
inc r_i ; r_i=r_i+1
rjmp loop ; go again
done:
```

Load Instruction (Absolute Addr)

LDS – Load Direct from Data Space

Description:
Loads one byte from the data space to a register.

A 16-bit address must be supplied. Memory access is limited to the current data segment of 64K bytes.

Operation:
(i)

Syntax:
(1) LDS Rd, k

Operands:
Rd = Rd < 16, k = k < 65536

Program Counter:
PC = PC + 2

32-bit Opcode:

<table>
<thead>
<tr>
<th>1991</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXX</td>
<td>XXXX</td>
<td>XXXX</td>
<td>XXXX</td>
</tr>
</tbody>
</table>

Note:
- 32-bit instruction with
- 16-bit immediate specifying an “absolute address”
- Recall memory SRAM is 1K words
### Store Instruction (Absolute Addr)

**STS – Store Direct to Data Space**

**Description:**
Stores one byte from a Register to the data space.
A 16-bit address must be supplied. Memory access is limited to the current data segment of 64K bytes.

**Operation:**
(i) \((k) \leftarrow Rr\)

**Syntax:**
- **Operands:** \(STS \; k;Rr\)
- **Program Counter:** \(PC \leftarrow PC + 2\)

**32-bit Opcode:**

```
1010 011d 4666 0000
```

**Note:**
- 32-bit instruction with 16-bit immediate specifying an “absolute address”
- Recall memory SRAM is 1K words

---

### Assembly Programming 301

- **E.g. High-level Code**

\[
A[8] = h + A[0]
\]

where \(A\) is an array of integers (1-byte each here)

- **Assembly Code**
  - suppose \(A\) is at location 0x100; \(h\) is \(r_h\)
  - suppose \(r_{temp}\) is a free register

```
lds r_temp, 0x100 ; r_temp = A[0]
add r_temp, r_h ; r_temp = h + A[0]
sts 0x108, r_temp ; A[8] = r_temp
```
Assembly Programming 302

- High-level Code
  \[
  \] (case 1)

  where \( A \) is an array of integers (1-byte each)

- Vs.
  \[
  \text{sum} = 0;
  \text{for} (i=0; i < 100; i=i+1)
  \text{sum} = \text{sum} + A[i];
  \] (case 2)

- Assembly Code
  - suppose \( A \) is at location 0x100; \( \text{sum} \) and \( i \) are in \( r_{sum}, r_i \)
  - suppose \( r_{temp} \) is a free register

Array Access in a Loop

- If memory address is hardcoded in the instruction, how do you read \( A[1] \) in the second iteration?
  - “self-modifying code”: load the instruction word; increment by 1; store it back
  - or, allow addresses to come from a data register where they can be manipulated like data
Load Instruction (Indirect Addr)

**LD – Load Indirect from Data Space to Register using Index X**

**Description:**
Loads one byte indirect from the data space to a register.
The data location is pointed to by the X (16 bits) Pointer Register in the Register File. Memory access is limited to the current data segment of 64K bytes.

**Syntax:**
LD Rd, X

**Operands:**
0 ≤ d ≤ 31

**16-bit Opcode:**

| (1) | 1001 | 000d | 088d | 090d |

**Comment:**
X: Unchanged

**Program Counter:**
PC ← PC + 1

**Note:**
- X is R26 (low byte) and R27 (high byte) viewed together as a 16-bit address
- LD supports two variants that perform arithmetic on X,
  - pre-increment LD Rd, X+ and post-decrement LD Rd, -X
- Also works with Y (R28,R29) and Z (R30, R31)

---

Store Instruction (Indirect Addr)

**ST – Store Indirect From Register to Data Space using Index X**

**Description:**
Stores one byte indirect from a register to data space.
The data location is pointed to by the X (16 bits) Pointer Register in the Register File. Memory access is limited to the current data segment of 64K bytes.

**Syntax:**
ST X, Fr

**Operands:**
0 ≤ r ≤ 31

**16-bit Opcode:**

| (1) | 1001 | 091e | xxxx | 110d |

**Comment:**

**Program Counter:**
PC ← PC + 1

**Program Counter:**
PC ← PC + 1

**Note:**
- Like LD, supports two variants that perform arithmetic on X,
  - pre-increment ST X+, Rd and post-decrement ST –X, Rd
- Also works with Y (R28,R29) and Z (R30, R31)
Addressing Modes in AVR

- **LDI Rd,k**: Immediate mode: put the constant k into Rd.
- **LDS Rd,k**: Absolute mode: use the constant k as an address to identify a memory location.
- **LD Rd,X**: Indirect mode: X or Y, or Z names a register pair.

**Register File**
- Data Memory
- Copy value from memory to Rd
- Copy value from memory to Rd
- Combine contents of two registers into a 16-bit address to identify a memory location

Assembly Programming 303

- **E.g. High-level Code**
  ```plaintext
  sum=0;
  for(i=0; i < 100; i=i+1)
    sum = sum + A[i];
  ```
  Where A is an array of integers (1-byte each)

- **Assembly Code**
  - Suppose A is at location 0x100; sum and i are in r_sum, r_i
E.g. High-level Code

```c
sum=0;
for(i=0; i < 100; i=i+1)
    sum = sum+ A[i];
```

where A is an array of integers (1-byte each)

Assembly Code

- suppose A is at location 0x100; sum and i are in r_sum, r_i

```assembly
ldi r26, 0x00
ldi r27, 0x01
ldi rsum, 0x0
ldi ri, 0x0

test: cpi ri, 100
brpl done
```

```assembly
ldi r26, 0x0100
ldi r27, 0x0100
ldi rsum, 0x0
ldi rupper, 0x0164

loop:
    ld rtemp, X+ ; auto-increment X
    add r_sum, r_temp
    cpi r26, 0x0164; i never calculated
    cpc r27, r_upper ; explicitly
    brne loop ; transformed while loop
```

High-level Code

```c
sum=0;
for(i=0; i < 100; i=i+1)
    sum = sum+ A[i];
```

Optimized Assembly Code

```assembly
ldi r26, lo8(0x0100)
ldi r27, hi8(0x0100)
ldi r_sum, 0x0
ldi r_upper, hi8(0x0164)

loop:
    ld r_temp, X+ ; auto-increment X
    add r_sum, r_temp
    cpi r26, lo8(0x0164); i never calculated
    cpc r27, r_upper ; explicitly
    brne loop ; transformed while loop
```
Is it better? By how much?

“Literal Version”

```
ldi r26,0x00 ; 1 cyc
ldi r27,0x01 ; 1 cyc
ldi r_sum, 0x00 ; 1 cyc
ldi r_i, 0x00 ; 1 cyc

test: cpi r_i, 100 ; 1 cyc
  brpl done ; 1 cyc
  ld r_temp, X ; 1 cyc
  add r_sum, r_temp ; 1 cyc
  inc r_i ; 1 cyc
  adiw r26, 1 ; 2 cyc
  rjmp test ; 2 cyc

done:
```

```
note: brpl=2 cyc if taken
```

 setTimeout; 1 cyc

```

ci metrics of goodness
- static inst count= 11
  (how many you see)
- dynamic inst count=
  4+100x7+2=706
  (how many executed)
- cycles=4+100x9+3=907

Atmel ISA manual specifies
effective delay in cycles
for the instructions
(simulator tells you too)
```

Is it better? By how much?

“Hacker Version”

```
ldi r26, lo8(0x100) ; 1 cyc
ldi r27, hi8(0x100) ; 1 cyc
ldi r_sum, 0x00 ; 1 cyc
ldi r_upper, hi8(0x164); 1 cyc

loop:
  ld r_temp, X+ ; 2 cyc
  add r_sum, r_temp ; 1 cyc
  cpi r26, lo8(0x164) ; 1 cyc
  cpc r27, r_upper ; 1 cyc
  brne loop ; 2 cyc
```

```
note: brpl=1 cyc if not taken
```

setTimeout; 1 cyc

```

• static inst count=9
  ~20% reduction
• dynamic inst count=
  4+100x5=504
  ~30% reduction
• cycles=4+100x7-1=703
  ~22% reduction

FYI, a good compiler will be
closer to Hacker than Literal
To Wrap up

- To be a hacker, you also need to know
  - subroutine calls, in particular recursive calls
  - exception handling
  - I/O
  - how to hand optimize code
  Feel free to read the AVR documents on Blackboard

- Big picture to keep in mind
  - you will see assembly programming in much greater detail in 213/240/34x/447 etc.
  - most of you will not code in assembly for a living; this is more about understanding the underlying concepts
  - once you learn one ISA you can learn the rest
    - some ISAs are uglier than others
    - AVR is not a pretty one.

Terminologies

- Instruction Set Architecture
  - the machine behavior as observable and controllable by the programmer
- Instruction Set
  - the set of commands understood by the computer
- Machine Code
  - instructions encoded in binary format
  - directly consumable by the hardware
- Assembly Code
  - instructions expressed in “textual” format
    - e.g. add r1, r2
    - converted to machine code by an assembler
    - one-to-one correspondence with machine code
      (mostly true: compound instructions, address labels ....)